Reasoning about Metamodeling with Formal Specifications and Automatic Proofs

Ethan K. Jackson[1], Tihamer Levendovszky[2], and Daniel Balasubramanian[2]

[1]Microsoft Research, Redmond, WA and [2]Vanderbilt University, Nashville, TN
ejackson@microsoft.com
{tihamer, daniel}@isis.vanderbilt.edu

Abstract. *Metamodeling* is foundational to many modeling frameworks, and so it is important to formalize and reason about it. Ideally, correctness proofs and test-case generation on the metamodeling framework should be automatic. However, it has yet to be shown that extensive automated reasoning on metamodeling frameworks can be achieved. In this paper we present one approach to this problem: Metamodeling frameworks are specified modularly using *algebraic data types* and *constraint logic programming* (CLP). Proofs and test-case generation are encoded as CLP satisfiability problems and automatically solved.

1 Introduction

Metamodeling is foundational to many modeling frameworks, and so it is important to formalize it properly. Ideally, a formalization should enable automated reasoning by generating test cases, proving correctness of the meta-interpreter, and proving correctness of editing operations. However, the state-of-the-art is somewhat less than ideal. On one hand, there has been a general consensus that the *Meta-Object Facility* (MOF) standard is under-formalized and deserves careful attention [1,2,3,4,5]. On the other hand, attempts at full formalization of MOF/MOF-alternatives have not yet enabled extensive automated reasoning on metamodeling frameworks. (We give a summary of existing results shortly.)

In this paper we present a new approach to formalizing and reasoning on metamodeling frameworks. The core of our approach uses *algebraic data types* (ADTs) and *constraint logic programming* (CLP) for formal specifications. We modularize these specifications so they mirror the key components of metamodeling frameworks: (1) A *model store* for representing models, metamodels, and conformance. (2) A set of model editing operations. (3) A *meta-interpreter* for promoting model-level elements to meta-level elements. We encode proof goals as instances of *CLP satisfiability* problems, and use our *FORMULA* framework to solve these instances [6]. The result is a concise formal specification whose structure resembles the tool architecture, but allows constructive automated reasoning to perform correctness proofs and test case generation.

Our contributions are: First, we develop a complete specification of a simple metamodeling framework based on *typed graphs* [7]. This gives a blueprint for

specifying more complicated frameworks. Second, We prove that our choice of editing operations preserves model conformance. We prove *metacircularity* by automatically constructing a meta-metamodel. These results have the interesting side-effect that it is unnecessary to write a bootstrapping meta-metamodel; it falls out of the proof. Third, we relate these results to a MOF-like metamodeling framework with richer conformance constraints, such as acyclicity and multiplicity constraints. These results are obtained using our FORMULA specification and analysis framework.

2 Related Work

Metamodeling continues to be an extensively researched topic [8] with many approaches to formalization. A few representatives are: The *Metamodeling Language Calculus* (MML) based on ς-calculus [1]. The graph-theoretic approaches of *KM3* [2] and *VPM* [3]. The work of [4] provides a rich set-theoretic setting for metamodeling. *MOMENT2* uses *membership equational logic* and *term rewriting* as a formal foundation [5]. Though formal, many approaches support limited automated reasoning on the metamodeling framework.

In this paper we investigate the power of automated formal methods to reason on metamodeling frameworks. Our tool supports expressive specifications corresponding to *fixpoint logic* (FPL) over theories [9], and provides a *finite model finder* for automated reasoning. Other automated techniques have been applied to metamodeling. The work of [5] uses *MAUDE* [10] to check metamodel/model conformance and *linear temporal logic* (LTL) properties via term rewriting systems. A proof is a reduction of an input term to a term with no further reductions. Model finding is not generally supported by this approach. Related to this, [11] describes translators from *VPM*-style specifications into explicit/symbolic state model checkers, though not necessarily for the purpose of reasoning on metamodeling frameworks.

The work of [12] provides a translation from UML and a subset of OCL into the finite model finder *Alloy* [13]. Alloy is perhaps the closest tool to FORMULA; for an extensive comparison see [6]. Alternative approaches avoid solving altogether, in favor of abstracting to graph grammars [14], or interactive theorem provers [15]. Yet, model finders continue to be effective for automative reasoning, even in areas such as *software product lines* [16].

These results are based on preliminary work we presented in [17]. To our knowledge, this is the first time such automated proofs have been shown for a metamodeling framework.

3 Introduction to CLP and Satisfiability

Constraint Logic programming (CLP) provides a powerful approach to writing formal specifications. First, a logic program Π can be directly (i.e. in polynomial-time) translated into first-order logic (FOL) according to its *Clark Completion*. Following the notation of [18], we refer to this translation as $\Pi^\star$. Second, logic

programs are executable, allowing programmatic reasoning to be applied while devising specifications. This form of reasoning is harder to obtain when directly writing FOL. Actually, an even stronger property holds: The execution of a logic program proves theorems about its logical semantics. If g is a quantifier-free formula over the relations computed by Π, then $\widetilde{\exists} g$ can be decided by executing Π. ($\widetilde{\exists} g$ denotes the existential closure of g.) The formula g is called a *goal*.

Consider the following program, which computes paths and cycles occurring in a directed graph.

Example 1 (Cycles).

$$\Pi_{cycles} \doteq \begin{array}{l} \mathsf{path(x,\ y)\ :\text{-}\ e(x,\ y).} \\ \mathsf{path(x,\ z)\ :\text{-}\ e(x,\ y),\ path(y,\ z).} \\ \mathsf{inCycle(x)\ :\text{-}\ path(x,\ x).} \end{array}$$

The symbols $e()$, $path(,)$, and $inCycle()$ are user defined relations. Each logic programming *rule* behaves like a universally quantified implication. Whenever the relations on the right-hand side of a rule hold for some substitution of the variables, then the left-hand side holds for that same substitution. A logic program is stronger than a set of implications, because it only entails theorems that can be explained by repeated applications of rules. Derivations must begin with *facts*, which are rules whose right-hand side is true. Formally, this means: (1) $\Pi^\star$ contains additional formulas to constrain the implications, and (2) the *intended interpretation* of $\Pi^\star$ is the smallest set of relations satisfying $\Pi^\star$. In this way, Example 1 encodes the transitive closure of a directed graph. (An alternative formalization for CLP is obtained by extending FOL with fixpoint operators [9].)

The program Π_{cycles} is not very interesting because it contains no facts. The least interpretation of this program assigns $e = path = inCycle = \emptyset$; it is called the *least Herbrand model* and denoted $lm(\Pi^\star)$. A goal $\widetilde{\exists} g$ holds for a program Π if g evaluates to true under the least Herbrand model; denoted:

$$lm(\Pi^\star) \models \widetilde{\exists} g.$$

In particular, $lm(\Pi^\star_{cycles}) \not\models \exists x\ inCycle(x)$. Suppose the program is extended with the fact $e(1,1)$, then exactly the additional facts $path(1,1)$ and $inCycle(1)$ are deducible and the goal is satisfied. Most LP languages are concerned with efficient rule application to prove a goal, either by working backwards from a goal to facts or forwards from facts to a goal.

3.1 CLP Satisfiability

We will generate automatic proofs from formal specifications by solving CLP *satisfiability* problems. Satisfiability is different from checking goal satisfaction; it is to determine if a program can be extended by a finite set of facts so that a goal is satisfied. As the previous example shows, this problem cannot be solved by simply running a logic program. It requires searching through (infinitely) many possible extensions, which we achieve by efficient forward *symbolic execution* of

a logic program into the state-of-the-art *satisfiability modulo theories* (SMT) solver *Z3* [19]. As a result, specifications can include variables ranging over infinite domains and rich data types. Nonetheless, the method is constructive; it returns extensions of the program witnessing goal satisfaction.

Let U be a (possibly infinite) set called a *universe* and r an n-ary relation symbol. Then a (finite) interpretation of r, written r^I, is a (finite) subset of U^n. We write $r(\vec{t})$ as a shorthand for r applied to elements $t_1, \ldots, t_n$ of U.

Definition 1 (CLP Satisfiability). *Given:*

1. *A program Π with relation symbols $R = \{r_1, r_2, \ldots, r_n\}$,*
2. *$R_p \subseteq R$ a subset of the program relations, called the primitive relations.*
3. *A quantifer-free goal g over the program relations.*

Then find a finite interpretation R_p^I for primitive relations such that:

$$lm((\Pi \cup R_p^I)^\star) \models \widetilde{\exists} g. \tag{1}$$

The program $\Pi \cup R_p^I$ is obtained by extending Π with a fact $r(\vec{t})$ whenever $R_P^I \models r(\vec{t})$.

The program can only be extended by primitive relations R_P. The contents of R_P^I are the facts that, when added to the program, cause the goal to be satisfied. We write $\mathbb{S}(\Pi, R_p, g)$ to denote an instance of CLP satisfiability and $R_P^I \in \mathbb{S}(\Pi, R_p, g)$ to denote an interpretation satisfying the problem. In a very technical sense, we refer to R_P^I as a *model* of $\mathbb{S}$. However, such interpretations can also represent instances of an abstraction, allowing them to serve as models in a more general sense. Thus, we may use the symbol M when more intuitive.

3.2 Blueprint of a Metamodeling Framework

We formalize metamodeling frameworks using CLP and CLP satisfiability according to the following blueprint:

1. **Model Store.** The model store encodes the set of all conforming metamodel/model pairs. It captures the semantics of metamodel conformance. Interesting instances of metamodel/model pairs can be constructed by solving satisfiability problems. We present a two-level model store, though an arbitrary number of meta-layers could be specified.
2. **Editing Operations.** These are transformations for editing model-level elements through creation and deletion. These transformations are also defined over the model store. By formalizing editors we can generate test cases where editing breaks model conformance. For a simple metamodeling framework we can choose model editors so that conformance is always maintained.
3. **Meta-interpreter.** The meta-interpreter is a transformation promoting model-level elements to meta-level elements. This transformation is defined over the model store. We say a framework is *metacircular* if there exists an input model that is promoted to its own metamodel by the meta-interpreter. Again, this property can be rephrased as a satisfiability problem and meta-metamodels are constructed witnessing this property.

4 Metamodeling by Typed Graphs

Typed graphs have been studied extensively as representations for (meta-) models, especially by the model transformation community [7]. For example, they are the basis for *KM3* metamodeling notation employed by the *ATLAS transformation language*, and can be used as a more basic foundation for MOF [2]. They are simple to define and easy to understand, so we use them to illustrate a complete metamodeling framework.

4.1 Typed Graphs and the Model Store

Definition 2 (Directed Graph). *A directed graph is a quadruple* $G = \langle V, E, src, dst\rangle$ *where* V *and* E *are sets;* $src : E \to V$ *and* $dst : E \to V$.

Definition 3 (Typed Graph). *A typed graph is a quadruple* $T = \langle G, H, \tau_v, \tau_e\rangle$ *where* G *and* H *are directed graphs;* $\tau_v : V_H \to V_G$ *and* $\tau_e : E_H \to E_G$.

The graph G acts like a metamodel providing a set of node types and edge types[1]. Graph H is an instance model referencing these types. The type of each vertex v is $\tau_v(v)$ and edge e is $\tau_e(e)$. A model H *conforms* to the metamodel G if the edges and vertices of H are connected according to their types:

$$conforms(T) \doteq \forall e \in E_H \begin{pmatrix} src_G(\tau_e(e)) = \tau_v(src_H(e)) \wedge \\ dst_G(\tau_e(e)) = \tau_v(dst_H(e)) \end{pmatrix}. \tag{2}$$

Fixing the universe U of edge/vertex labels yields a set of all possible conforming metamodel/model pairs. We call this set the *model store*:

$$Store(U) \doteq \{T \mid conforms(T) \ \wedge \ V_G, E_G, V_H, E_H \subseteq U\}. \tag{3}$$

4.2 Specifying the Model Store with ADTs and CLP

Figure 1 shows an equivalent specification of the model store in FORMULA. This specification is wrapped in a ***domain*** block, which delimits a domain-specific abstraction. As mentioned earlier, FORMULA directly supports algebraic data types and these are used to encode user defined relations. For example, Line 3 declares a data type constructor *MetaNode*() for instantiating meta-level nodes (V_G). This constructor produces *MetaNode* records, each of which has a field called *typename* of type *String*. Similarly, *MetaEdge*(, ,) constructs elements of E_G using *MetaNode*s as endpoints (Line 5). The *Node* and *Edge* constructors instantiate model-level elements (graph H), and the fields called *type* encode τ_v and τ_e.

Due to the flexibility of ADTs, it is unnecessary to distinguish between data type constructors and user-defined relation symbols. Instead, every program computes two standard unary program relations, r_p and r_d, over records.

[1] Our definition differs from others as we allow edges to also acts as types.

```
1.  domain ModelStore
2.  {
3.    MetaNode  ::= (typename: String).
4.    [Closed(src, dst)][Unique(typename -> src, dst)]
5.    MetaEdge  ::= (typename: String, src: MetaNode, dst: MetaNode).
6.    [Closed(type)][Unique(name -> type)]
7.    Node      ::= (name: String, type: MetaNode).
8.    [Closed(src, dst, type)][Unique(name -> src, dst, type)]
9.    Edge      ::= (name: String, src: Node, dst: Node, type: MetaEdge).
10.
11.   badSrc    :=Edge(_, src, dst, t), t.src != src.type.
12.   badDst    :=Edge(_, src, dst, t), t.dst != dst.type.
13.   conforms  :=!badSrc & !badDst.
14. }
```

Fig. 1. FORMULA specification of a model store containing typed graphs.

The primitive relation r_p contains only records built with *primitive constructors*, and the derived relation r_d contains only records built with *derived constructors*. Primitive constructors can be used to extend a program in order to solve a satisfiability problem; derived constructors cannot. Primitive constructors always begin with a capital letter. Every FORMULA domain contains a special nullary derived constructor called ***conforms***. The models of a domain D are those extensions of Π by r_p where *conforms* is derivable:

$$models(D) \doteq \{r_p^I \mid r_p^I \in \mathbb{S}(\Pi_D, \{r_p\}, conforms)\}. \tag{4}$$

FORMULA provides special syntax for expressing domain models, as shown in Figure 2. The declaration ***model*** M ***of*** D is a claim that the code-to-follow gives an interpretation $r_P^I \in models(D)$. This claim is checked by the compiler. Recall that r_p^I is just a set of records, thus a *model* block is just a set of records. The *StateDiagram* model in Figure 2 is an instance of the model store representing a small state diagram over the meta-types *State* and *Transition*.

The constraints describing typed graph conformance are expressed in Lines 11 - 13 of Figure 1. FORMULA also provides special syntax for rules where the left-hand side is a nullary constructor. We refer to these as *queries* and use the *query definition* operator (:=) for query definitions. Intuitively, a query behaves like a propositional variable that is true if and only if the right-hand side of the definition is true for some substitution. As a convenience, FORMULA allows queries to be treated like propositional variables when they appear in other query definitions. For example, the *badSrc* query in Line 11 of Figure 1 detects if the source of a model-level edge has been connected improperly. It corresponds to the following formula in $\Pi^\star$:

$$badSrc \in r_d \Leftrightarrow \exists Edge(n, src, dst, t) \in r_p \; get_{src}(t) \neq get_{type}(src). \tag{5}$$

```
1.  model StateDiagram of ModelStore
2.  {
3.    MetaNode("State")
4.    MetaEdge("Transition",
5.                MetaNode("State"),
6.                MetaNode("State"))
7.    Node("S1", MetaNode("State"))
8.    Node("S2", MetaNode("State"))
9.    Edge("T1",
10.         Node("S1", MetaNode("State"))
11.         Node("S2", MetaNode("State"))
12.         MetaEdge("Transition",
13.                  MetaNode("State"),
14.                  MetaNode("State")))
15. }
```

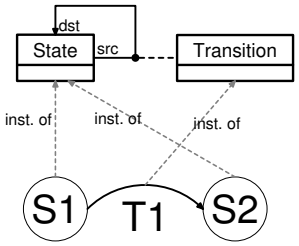

Fig. 2. A FORMULA model from the model store that encodes a state diagram.

where $get_x()$ extracts the field named x. Similarly, the *conforms* query is expressed as:

$$conforms \in r_d \Leftrightarrow badSrc \notin r_d \wedge badDst \notin r_d \wedge \varphi_{compiler}. \tag{6}$$

The sub-formula $\varphi_{compiler}$ holds additional conformance constraints that are automatically added by the compiler. These extra constraints may appear due to inheritance of constraints through the module system or due to shorthands. One such shorthand is the *Closed* annotation (Line 4), which requires the *src*/*dst* fields of *MetaEdge* to contain only meta-nodes declared at the top level. The *Unique* annotation requires all records with identical fields on the left of the arrow (–>) to have identical fields on the right of the arrow. These shorthands encompass many common constraints, though it always possible to express the same constraints without using them.

In summary, the FORMULA specification encodes a typed graph model store using ADTs and CLP. The set of all conforming metamodel/models pairs is characterized by the satisfiability problem *models*(*ModelStore*). Therefore, we can use automated techniques to prove properties about the model store. We shall illustrate this in the later sections. But first, our typed graph framework requires a few operations for editing models: Delete node, create node, delete edge, and create edge. In the next sections we show how to specify these operations, illustrating that CLP satisfiability can also be used to reason about model transformations.

5 Encoding Model Transformations

Model transformations are encoded as logic programs where data types distinguish the inputs and outputs of the transformation. For example:

Example 2 (Filter MetaNodes).

$$\Pi_{filter} \doteq \mathsf{out.MetaNode(x)} \text{ :- } \mathsf{in.MetaNode(x)}.$$

The constructor *in.MetaNode*() stands for meta-node primitives at the input of the transformation. Similarly, *out.MetaNode*() stands for meta-node primitives on the output of the transformation. A transformation is executed by providing an interpretation r_p^I for the input primitives, and then computing the output primitives according to the logic program:

$$transform(\Pi, r_p^I) \doteq \{f(\vec{t}\,) \mid lm((\Pi \cup r_p^I)^\star) \models f(\vec{t}\,) \wedge isOut(f)\} \tag{7}$$

where the predicate $isOut(f)$ tests if constructor f is an output primitive.

In order to ease the use of transformations we introduce the *renaming operator* ***as***. Let Π ***as*** X return a new program Π_X obtained by replacing every occurrence of a function symbol f with $X.f$ in Π. This also applies to the type declarations in Π. Similarly, r_P^I ***as*** X replaces every f-record with an equivalent $X.f$ record. Thus, the program Π_{filter} can be used to transform the model *StateDiagram* in Figure 2 as follows:

$$transform(\Pi_{filter}, StateDiagram \textbf{ as } in) = \{out.MetaNode(\text{“State”})\}.$$

The *filter* transformation only copies meta-nodes to the output, so it effectively deletes all other information from the output.

Satisfiability can be used to reason about model transformations. The approach is to compose renamed versions of input/output domains with the transformation in order to reason about its impact on domain constraints. For example, we may wish to know if there exists a conforming instance from the model store that is no longer conforming after *filter* is applied.

Example 3 (Property Conformance-Breaking).

$$\Pi_{CB} \doteq \Pi_{filter} \cup (\mathit{ModelStore} \textbf{ as } \mathit{in}) \cup (\mathit{ModelStore} \textbf{ as } \mathit{out}) \cup \mathsf{confBreaking} := \mathsf{in.conforms} \;\&\; \mathsf{!out.conforms}.$$

The problem $\mathbb{S}(\Pi_{CB}, \{r_p\}, confBreaking)$ has a solution if and only if there exists such an input to the transformation. (Note that r_p only contains input primitives; output primitives are placed in r_d). In this case, the problem is unsatisfiable, so there is no such input to the transformation.

5.1 Editing By Transformations

The FORMULA module system simplifies the specification of model transformations, as shown in Figure 3. Line 1 declares the transformation called *CreateNode*, which requires two *parameters* called *newName* and *newType*. Parameters are extra pieces of information provided to the transformation, in addition to the input models. These parameters give the name and type of the node to be added. Line 2 identifies the inputs/outputs of the transformation by two lists of renamed

domains. The compiler automatically composes the transformation logic (Lines 4 - 10) with the renamed input/output domains.

Lines 4 - 7 of the transformation copy the input metamodel/model to the output of the transformation. These rules are particularly simple due to renaming inference by the compiler. For example, the right-hand side of the rule in Line 5 has a variable called *src* that must be of type *in.MetaNode*. However, the constructor *out.MetaNode* on the left-hand side expects *src* to be of type *out.MetaNode*. The compiler detects this and applies renaming to *src* on the left-hand side. In addition to copying, Line 8 adds a new node called *newName* to the output if such a node does not already exist. Line 10 specifies the conformance-breaking property.

The *CreateNode* transformation has all the context needed for proving properties about its behavior. The solver can be used to find an instance of the inputs and parameters causing conformance to be broken. Because there are many degrees of freedom in this problem, it is useful to give the solver some guidance. We call this guidance a *partial model*; it is roughly a lower bound on the structure of r_p^I. Figure 4 shows a partial model containing three applications of each primitive constructor to *fresh variables* (denoted _). This partial model requires the solver to return models with at least one (meta-)node/edge each. In addition, the free variables cause the solver to eagerly search for larger models. The size and structure of r_p^I may be further expanded during the search process, beyond the contents of the partial model. In order to check the conformance-breaking property, we issue the following command to FORMULA:

```
solve CreateNode <_,_> PInst confBreaking
```

This allows the solver to search for any parameter values that break conformance when applied to some instance of the model store. In this case, the problem is satisfiable because an undeclared meta-node may be provided as the *newType* parameter. (This violates the *Closed* constraint in Figure 1, Line 6.) Figure 5 shows an example of the FORMULA output, which consists of a model and parameter valuations solving the satisfiability problem.

```
1.  transform CreateNode  <newName : String, newType : in.MetaNode>
2.  from ModelStore as in to ModelStore as out
3.  {
4.    out.MetaNode(typename)             :-  in.MetaNode(typename).
5.    out.MetaEdge(typename, src, dst)   :-  in.MetaEdge(typename, src, dst).
6.    out.Node(name, type)               :-  in.Node(name, type).
7.    out.Edge(name, src, dst, type)     :-  in.Edge(name, src, dst, type).
8.    out.Node(newName, newType)         :-  fail in.Node(newName, _).
9.
10.   confBreaking    := in.conforms & !out.conforms.
11. }
```

Fig. 3. A *CreateNode* transformation in FORMULA.

```
1.  partial model PInst of ModelStore
2.  {
3.    MetaNode(_) MetaNode(_) MetaNode(_)
4.    MetaEdge(_, _, _) MetaEdge(_, _, _) MetaEdge(_, _, _)
5.    Node(_, _) Node(_, _) Node(_, _)
6.    Edge(_, _, _, _) Edge(_, _, _, _) Edge(_, _, _, _)
7.  }
```

Fig. 4. Partial instance of r_p^I to guide the solver.

```
1.  model Proof of ModelStore
2.  {
3.    MetaNode("A")
4.    MetaEdge("B", MetaNode("A"),  MetaNode("A"))
5.    Node("C", MetaNode("A"))
6.    Edge("D", Node("C", MetaNode("A")), Node("C", MetaNode("A")),
7.           MetaEdge("B", MetaNode("A"), MetaNode("A")))
8.  }
9.  newName = "E", newType = in.MetaNode("F")
```

Fig. 5. An automatically generated witness that *CreateNode* is conformance breaking.

5.2 Conformance-Preserving Edits

The typed graph formalism is simple enough that we can define editing operations which never break conformance. Specifically, *CreateNode* should only create a node if there is no other node with same name and the meta-node exists. Thus, Line 8 is replaced by:

```
out.Node(newName, newType)  :-  in.MetaNode(n), n = newType.typename,
                                fail in.Node(newName, _).
```

Similar rules hold for *CreateEdge*. *DeleteNode* must also delete all incident edges. See `http://research.microsoft.com/formula` for the complete specification of these transformations.

In general it is undecidable whether or not there exists a finite interpretation satisfying a CLP satisfiability problem. Therefore, the solver can only guarantee the absence of solutions up to some size of r_p^I. This is a well-known problem when using constructive methods to generate proofs. Fortunately, there is a well-known solution: Provide an inductive argument that generalizes the absence of solutions to interpretations of arbitrary size. The advantage of such inductive arguments is that they can be rather generic and reusable across problem instances.

For example, the FORMULA solver can be used to show that for all conforming inputs r_p^I where $|r_p^I| \leq k$, then no editing operation breaks conformance. These results can be paired with a theorem showing that all other cases can be decomposed into these small cases. First, a *term homomorphism* φ is a function from records to records with the property that $\varphi(f(t_1, \ldots, t_n)) = f(\varphi(t_1), \ldots, \varphi(t_n))$.

Let the *base cases* $\mathbb{B}$ be a finite set of input interpretations, and $\tau\langle\vec{x}\rangle(M)$ denote an editing operation with parameters $\vec{x}$ applied to model (input interpretation) M.

Theorem 1 (Decomposition Theorem). *The transformation $\tau\langle\vec{x}\rangle(M)$ is equivalent to transforming a relabeled instance of $\mathbb{B}$ and combining it with a subset of M. In symbols:*

$$\forall M, \vec{x} \;\; \exists M', M'', \varphi \quad \tau\langle\vec{x}\rangle(M) = M' \cup \varphi^{-1}\Big(\tau\big\langle\varphi(\vec{x})\big\rangle\big(\varphi(M'')\big)\Big) \tag{8}$$

such that:

$$M = M' \cup M'' \quad \textit{and} \quad \exists B \in \mathbb{B}\ \varphi(M'') = B. \tag{9}$$

Note, $\varphi(M)$ is φ applied to every term in M. The function φ^{-1} is the inverse image of φ.

This theorem formalizes the fact that an edit operation acts locally on an input. Reasoning on the set of base cases $\mathbb{B}$ is sufficient, because every input can be described as a local edit on a base case. In fact, $\mathbb{B}$ need not be constructed manually; only an upper bound k on the largest interpretation in $\mathbb{B}$ needs to be constructed. While the proof of this theorem is not automatic, its form is not specific to this example so it provides a general proof strategy.

6 The Meta-interpreter

A meta-interpreter is a transformation promoting model-level elements to meta-level elements. When combined with editing operations, it provides a way to build new abstractions using the operations provided by the framework.

Figure 6 shows one such meta-interpreter in FORMULA; there are several noteworthy aspects. First, the promotion is determined by arbitrary and hard-coded type names. Lines 3, 4 promote a model-level node to a meta-node only if the node has type *Class*. Similarly, the promotion of edges to meta-edges only occurs if an edge's type is *Assoc* and its end-points are *Class*es (Lines 5 - 11). Second, the choice of type names is unrelated to the formalization of the model store. The strings "Class" and "Assoc" are convenient, but arbitrary, monikers. Thus, the model store may have a simpler formalization than the concepts exposed by the meta-interpreter (though perhaps less convenient). Certainly, the model store can be insulated from the naming of the concepts, which may vary between standards.

Metamodeling frameworks are said to be bootstrapped by a *meta-metamodel* or are *"described using themselves"* [20]. Informally, a framework is *meta-circular* if there exists a metamodel *MM* whose conforming models are metamodels and *MM* is among them. Of course, this terminology has concerned many researchers, as it may lead to circular definitions. We formalize meta-circularity as a simple property of the framework:

```
1.  transform MetaInterpreter from ModelStore as in to ModelStore as out
2.  {
3.    out.MetaNode(name)     :-  in.Node(name, type),
4.                               type = in.MetaNode("Class").
5.    out.MetaEdge(name,
6.       MetaNode(srcname),
7.       MetaNode(dstname))  :-  in.Edge(name, src, dst, type),
8.                               src = in.Node(srcname, mClass),
9.                               dst = in.Node(dstname, mClass),
10.                              type = in.MetaEdge("Assoc", mClass, mClass),
11.                              mClass = in.MetaNode("Class").
12. }
```

Fig. 6. A simple meta-interpreter that promotes nodes of type *Class* and edges of type *Assoc* to the meta-level.

```
1.  metaDiffers    := in.MetaNode(t), fail out.MetaNode(t).
2.  metaDiffers    := out.MetaNode(t), fail in.MetaNode(t).
3.  metaDiffers    := in.MetaEdge(t, MetaNode(st), MetaNode(dt)),
4.                    fail out.MetaEdge(t, MetaNode(st), MetaNode(dt)).
5.  metaDiffers    := out.MetaEdge(t, MetaNode(st), MetaNode(dt)),
6.                    fail in.MetaEdge(t, MetaNode(st), MetaNode(dt)).
7.  metaCircular   := !metaDiffers & in.conforms & out.conforms.
```

Fig. 7. Specification of meta-circularity for the typed graph framework.

Definition 4 (Meta-circularity). *A framework is a meta-circular if there exists a conforming input to the meta-interpreter producing a conforming output with the same metamodel.*

The input witnessing this property is the meta-metamodel.

In our approach neither meta-circularity is required for bootstrapping nor does a meta-metamodel determine properties of the framework. Instead, the framework is determined by the model store, editing operations, and meta-interpreter. A meta-metamodel, if it exists, is a byproduct of this framework. In fact, it can be constructed automatically as a witness to the meta-circularity property. Figure 7 shows the specification of meta-circularity in FORMULA. The query definitions in Lines 1 - 2 test if there exists a meta-node in the input, which is not in the output, and *vice versa*. Lines 3 - 6 perform the same test for meta-edges. Then meta-circularity is simply the absence of any discrepancies at the meta-level of the input and output, both of which must conform to the model store.

A meta-metamodel is constructed by adding the specification of meta-circularity to the meta-interpreter and invoking the solver as follows:

```
solve MetaInterpreter PInst metaCircular
```

The result is the meta-metamodel of Figure 8. There is an additional use for this meta-metamodel; it provides a starting point for building metamodels using

```
1.  model MetaMetaModel of ModelStore
2.  {
3.    MetaNode("Class")
4.    MetaEdge("Assoc", MetaNode("Class"),  MetaNode("Class"))
5.    Node("Class", MetaNode("Class"))
6.    Edge("Assoc",
7.          Node("Class", MetaNode("Class")),
8.          Node("Class", MetaNode("Class")),
9.          MetaEdge("Assoc", MetaNode("Class"), MetaNode("Class")))
10. }
```

Fig. 8. An automatically generated meta-metamodel witnessing meta-circularity.

only the framework operations. For example, the state diagram abstraction used in Figure 2 can be constructed as follows:

$$\left(\begin{array}{c} \tau_{mi} \circ \\ \tau_{+edge} \left\langle \begin{array}{c} \textit{“Transition”}, \textit{in.Node}(\textit{“State”}), \textit{in.Node}(\textit{“State”}), \\ \textit{in.MetaEdge}(\textit{“Assoc”}, \ldots) \end{array} \right\rangle \circ \\ \tau_{+node} \langle \textit{“State”}, \textit{in.MetaNode}(\textit{“Class”}) \rangle \circ \\ \tau_{mi} \end{array} \right) MM \quad (10)$$

where MM is the meta-metamodel, τ_{mi} is an application of the meta-interpreter, τ_{+edge} creates an edge, and $\tau_2 \circ \tau_1$ is the application of τ_2 after τ_1. If the semantics of the model store or meta-interpreter are changed, then the starting point MM can be automatically reconstructed. To our knowledge, this is the first time such a technique has been demonstrated.

7 A MOF-like Framework

Several issues arise when specifying a richer metamodeling framework, such as the *Meta-Object Facility* (MOF). The first issue is the number of additional concepts that must be specified. Naturally, this is handled by introducing more types in the model store and more rules in the meta-interpreter. In the case of MOF, the key concepts at the meta-level are *Classifier*, *Class*, *Association*, *Generalization*, and *Property*. At the instance level there are *InstanceSpecification*, *InstanceValue*, and *Slot* concepts. An instance is related to one or more meta-level *Classifier*s, which include *Class*es and *Association*s. Each instance specification contains *Slot*s, which bind *Value*s to *Property*. The endpoints of n-ary associations are expressed using slots and properties.

The second issue is the expressiveness needed to define the model store. Here the primary complications are acyclicity and multiplicity constraints. MOF requires the generalization relationship to be acyclic and strong containment to be tree-structured. These constraints are not first-order definable, as they are equivalent to finite transitive closure. Fortunately, CLP exposes fixpoint operators via recursive rules (see Example 1), so acyclicity constraints are easily

captured. Multiplicity constraints require the number of instances related to another to be in an interval $[k_l, k_u]$. FORMULA supports encoding of multiplicity constraints through *aggregation operators*, such as *count*(), which count the number of facts matching some pattern. For example, the following rule:

```
outEdgesInInterval(n)  :-  n is Node, Multiplicity(n, kl, ku),
                           count(Edge(_, n, _, _)) >= kl,
                           count(Edge(_, n, _, _)) <= ku.
```

produces an *outEdgesInInterval(n)* fact for every node n whose out-edges number between $k_l(n)$ and $k_u(n)$. The expression n ***is*** *Node* is a shorthand for *Node*(_, _). Please see `http://research.microsoft.com/formula` for an example of a MOF-like framework.

8 Discussion and Conclusion

We have provided a modular specification of a complete metamodeling framework using ADTs and CLP. The key components of this specification where: (1) a model store, formalizing the legal metamodel/model pairs, (2) editing operations, formalizing the evolution of the model-level elements within the framework, (3) a meta-interpreter, formalizing the promotion of elements from the model-level to the meta-level. We have illustrated that FORMULA simplifies the presentation through the use of *domains*, *transformations*, and *partial models*. We have shown that proofs can be phrased as CLP satisfiability problems and automatically solved. Using this approach we were able to provably synthesize the meta-metamodel of the specified framework. To our knowledge, this is the first time this has been accomplished. It also shows concretely that meta-metamodels simply fall out of the specification, and are not paradoxical. (Though this fact has long been known.)

Throughout this paper we focused on automatic proofs, though test-case generation is another immediate consequence. This can be accomplished by describing a regime of interesting test-cases using a query definition, and then constructing instances satisfying the query. For example, to generate metamodels we solve for conforming instances of the model store with no model-level elements. To generate models conforming to a metamodel, we solve for conforming instances that share a common fixed meta-level. The FORMULA module system makes it straightforward to add these additional constraints for the purpose of test-case generation.

These results point the way to interesting future work. First, there is the question of how to automatically generalize unsatisfiability results to interpretations of arbitrary size. Positive theoretical results include known fragments of CLP that are decidable and well-behaved. However, we do not know of existing tools that leverage these results to compute an automatic upper-bound on the size of r_p^I. Second, there are other properties of a metamodeling framework that might be of interest. We might want to know a *closure property* that every well-formed instance of the model store can be constructed by starting from the

meta-metamodel and applying the framework operations. Automatically deciding such a property may very well require symbolic model checking in addition to the techniques illustrated here.

References

1. Clark, T., Evans, A., Kent, S.: The Metamodelling Language Calculus: Foundation Semantics for UML. In: FASE. (2001) 17–31
2. Jouault, F., Bézivin, J.: KM3: A DSL for Metamodel Specification. In: FMOODS. (2006) 171–185
3. Varró, D., Pataricza, A.: VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Journal of Software and Systems Modeling **2**(3) (October 2003) 187–210
4. Alanen, M., Porres, I.: A Metamodeling Language Supporting Subset and Union Properties. Software and System Modeling **7**(1) (2008) 103–124
5. Boronat, A., Meseguer, J.: An Algebraic Semantics for MOF. Formal Asp. Comput. **22**(3-4) (2010) 269–296
6. Jackson, E.K., Kang, E., Dahlweid, M., Seifert, D., Santen, T.: Components, platforms and possibilities: towards generic automation for MDA. In: EMSOFT. (2010) 39–48
7. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science). An EATCS Series. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)
8. Atkinson, C., Kühne, T.: The Essence of Multilevel Metamodeling. In: UML. (2001) 19–33
9. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Comput. Surv. **33**(3) (2001) 374–425
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and Programming in Rewriting Logic. Theor. Comput. Sci. **285**(2) (2002) 187–243
11. Varró, D.: Automated Formal Verification of Visual Modeling Languages by Model Checking. Software and System Modeling **3**(2) (2004) 85–113
12. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A Challenging Model Transformation. In: MODELS. (2007) 436–450
13. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: TACAS. (2007) 632–647
14. Ehrig, K., Küster, J.M., Taentzer, G.: Generating Instance Models From Meta Models. Software and System Modeling **8**(4) (2009) 479–500
15. Grönniger, H., Ringert, J.O., Rumpe, B.: System Model-Based Definition of Modeling Language Semantics. In: FMOODS/FORTE. (2009) 152–166
16. Mendonça, M., Wasowski, A., Czarnecki, K.: SAT-based Analysis of Feature Models is Easy. In: SPLC. (2009) 231–240
17. Jackson, E.K., Sztipanovits, J.: Constructive Techniques for Meta- and Model-Level Reasoning. In: MODELS. (2007) 405–419
18. Jaffar, J., Maher, M.J., Marriott, K., Stuckey, P.J.: The Semantics of Constraint Logic Programs. J. Log. Program. **37**(1-3) (1998) 1–46
19. de Moura, L.M., Bjørner, N.: Z3: An efficient smt solver. In: TACAS. (2008) 337–340
20. Object Management Group: Meta Object Facility (MOF) Core Specification Version 2.4. (2010)